Bahnunternehmen

Aufgaben

Ein privates Bahnunternehmen betreibt auf ausgewählten Strecken des deutschen Schienennetzes überregionalen Bahnverkehr zwischen verschiedenen Großstädten. Benachbarte Bahnhöfe werden jeweils durch Streckensegmente bestimmter Längen miteinander verbunden. Organisation und Verwaltung des Bahnverkehrs werden mit einem Datenbanksystem und einem objektorientierten Softwaresystem durchgeführt.
Material 1 zeigt eine schematische Übersicht des Streckennetzes.

1 Das Bahnunternehmen verwaltet Netz- und Fahrplandaten in einer relationalen Datenbank. Der Zugverkehr auf dem Streckennetz ist in Bahnlinien organisiert. In einem Fahrplan sind Ankunft- und Abfahrtzeiten der Züge für jeden Bahnhof festgelegt. Alle Züge einer Linie fahren im 1- oder 2-Stunden Takt. Beispielsweise fährt stündlich ein Zug von Hamburg nach Freiburg, um 6:45, um 7:45, um 8:45 usw. (Material 2). Die Haltezeiten der Züge sind bahnhofsabhängig. Beispielsweise halten alle Züge in Frankfurt 5 min, während sie in Berlin 10 min halten.

1.1 Ein erstes Entity-Relationship-Modell A (ERM A) wird durch das Diagramm in Material 3 dargestellt. Überführen Sie das ERM A in ein relationales Modell in 3. Normalform und erläutern Sie Ihre Lösung.

Hinweis: Alle Relationen sind in der Schreibweise `Relation(`<u>`PK`</u>`, Attribut, ..., FK#)` anzugeben.

(8 BE)

1.2 Das Unternehmen plant, seine Fahrpläne zukünftig den Jahreszeiten anzupassen und einen Sommer- und einen Winterfahrplan zu betreiben. Beim jahreszeitlichen Fahrplanwechsel werden die Abfahrt- und Ankunftszeiten vieler Züge verändert. Das Konzept des getakteten Linienbetriebs wird beibehalten. Das Unternehmen hat alternativ zum ERM A (Material 3) das Entity-Relationship-Modell B (ERM B) entwickelt (Material 4). Analysieren und diskutieren Sie die beiden Modelle unter den Aspekten des Ressourcenbedarfs und des jahreszeitlichen Fahrplanwechsels.

(6 BE)

1.3 Die relationale Datenbank wurde auf Grundlage des ERM B entwickelt. Das Relationenmodell ist in Material 5 dargestellt.

1.3.1 Geben Sie eine SQL-Anweisung an, die das längste Segment und die Gesamtlänge des Streckennetzes ermittelt.

(2 BE)

1.3.2 Geben Sie eine SQL-Anweisung an, die den Zug ICE 944 aus der Datenbank entfernt.

(2 BE)

1.3.3 Die Linien 26A und 26B werden um einen neuen Streckenabschnitt zwischen Freiburg und Basel erweitert (Material 1). Der Bahnhof in Basel (BA) wird neu in das Bahnnetz aufgenommen. Dort halten alle Züge 5 Minuten. Die Entfernung zwischen den beiden Bahnhöfen beträgt 70 km, die Höchstgeschwindigkeit für Züge auf diesem Teilstück liegt bei 250 km/h. Die Fahrzeit beträgt 35 Minuten. Alle Züge halten in Freiburg 5 Minuten. Züge der Linie 26B werden in Basel jeweils um XX:07 Uhr in Richtung Hamburg abfahren.
Geben Sie SQL-Anweisungen an, die die neuen Informationen in die Datenbank aufnehmen, und implementieren Sie die erforderlichen SQL-Anweisungen zum Aktualisieren der Datenbank.

Hinweise: Für die Verbindung zwischen zwei Bahnhöfen wird jeweils ein Streckensegment für jede Richtung angelegt. Den numerischen Primärschlüssel der Relation `Segment` erzeugt die Datenbank automatisch. Beachten Sie die in Material 1 aufgeführten Positionsnummern.

(11 BE)

1.3.4 Das Unternehmen stellt für die Bahnhöfe seines Netzes chronologisch sortierte Abfahrtspläne bereit, die für alle Züge die Linie, den Zug, die Abfahrtszeit und den nächsten Bahnhof (Material 1) angeben. Der Bahnhof in München ist ausschließlich Start- und Endpunkt von Linien.
Implementieren Sie die SQL-Anweisung, die den Abfahrtsplan für den Bahnhof in München ausgibt.

Hinweis: Die Funktion `MAKETIME(h, min, s)` erstellt aus den übergebenen Werten für Stunden (`h`), Minuten (`min`) und Sekunden (`s`) einen Zeitwert im Format `hh:mm:ss`.

(5 BE)

1.4 Das Bahnunternehmen möchte die Datenbank aus dem ERM B erweitern, um Einsatzzeiten seiner Triebwagen und Waggons zu erfassen. Dazu sind folgende neue Anforderungen von der Datenbank zu erfüllen:

- Jeder Zug besteht aus zwei Triebwagen. Ein Triebwagen kann für unterschiedliche Züge genutzt werden.
- Zwischen den Triebwagen sind mehrere Waggons eingekoppelt.
- Der Waggonpark des Unternehmens umfasst Großraumwaggons, Abteilwaggons und Restaurantwaggons.
- Ein Waggon ist für die erste oder die zweite Klasse gebaut.
- Sowohl für Triebwagen als auch für Waggons sind das Datum der Erstinbetriebnahme und die Einsatztage zu speichern.

Entwickeln und zeichnen Sie ein Entity-Relationship-Modell für die neuen Anforderungen.

Hinweise: Es sind nur die Entitätstypen in Ihr Modell zu übernehmen, die für die Modellierung der neuen Anforderungen erforderlich sind. Die Erweiterungen sind mit Entitätstypen, Attributen und Beziehungen sowie deren Kardinalitäten in [min, max]-Notation darzustellen.

(6 BE)

2 Das Streckennetz wird in einer objektorientierten Software abgebildet. Das Netz besteht aus Bahnhöfen, die durch Streckensegmente miteinander verbunden sind.

2.1 Das Bahnunternehmen bietet die Möglichkeit, mithilfe einer Routenfindersoftware Bahnreisen zu planen. In Material 6 finden Sie einen Ausschnitt des UML-Klassenmodells der Software.

2.1.1 Nennen Sie die im UML-Klassendiagramm auftretenden Beziehungstypen.
Erläutern Sie die Bedeutung jedes Beziehungstyps anhand eines Beispiels aus dem vorliegenden Diagramm sowie die Implementierung von Beziehungen in einer objektorientierten Programmiersprache.

(6 BE)

2.1.2 Überführen Sie die im UML-Klassendiagramm dargestellte Klasse `Bahnhof` in Anweisungen einer objektorientierten Programmiersprache und implementieren Sie die Methoden.

Hinweis: Die Dokumentation der Klasse `List` ist in Material 7 zu finden.

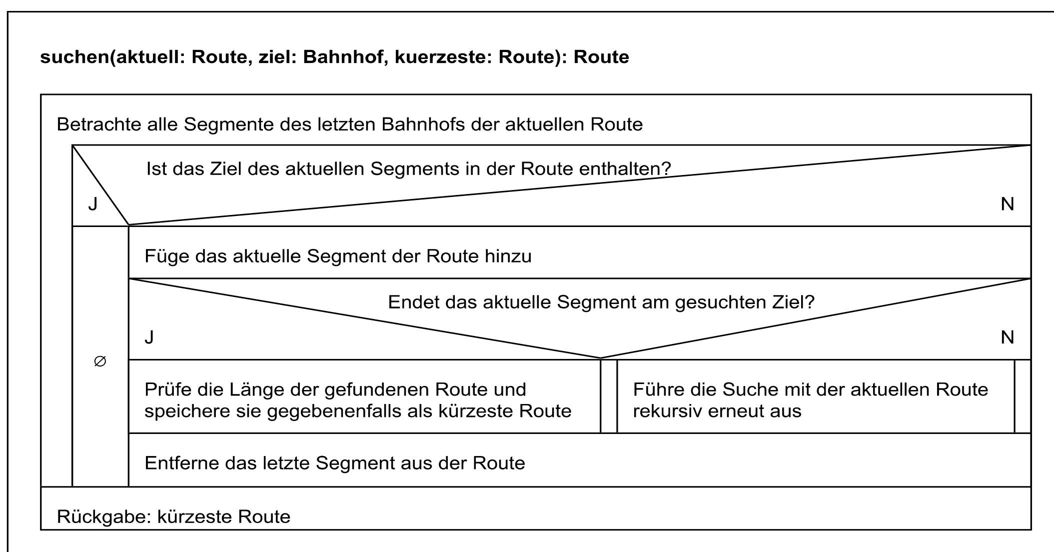
(4 BE)

2.1.3 Die Klasse `Streckennetz` (Material 6) stellt mit der Methode `findeKuerzesteRoute(start: String, ziel: String)` die öffentliche Schnittstelle für die Routensuche zwischen zwei Bahnhöfen bereit. Wenn die Parameter `start` und `ziel` gültige und unterschiedliche Bahnhofsnamen enthalten, ermittelt sie mithilfe der Methode `suchen()` die kürzeste Route.
Implementieren Sie die Klasse `Streckennetz` und die Methode `findeKuerzesteRoute()`.

Hinweise: Beim ersten Aufruf der Methode `suchen()` wird für die kürzeste Route (Parameter `kuerzeste`) `null` übergeben. Die Dokumentation der Klasse `List` ist in Material 7 zu finden.

(5 BE)

2.1.4 Die Methode `suchen()` der Klasse `Streckennetz` (Material 6) verwendet den nachfolgend beschriebenen Algorithmus zur Suche nach der kürzesten Route zwischen zwei Bahnhöfen. Falls keine Route gefunden wird, gibt die Methode `null` zurück.



Implementieren Sie die Methode `suchen()`.

(5 BE)

2.2 Das Bahnunternehmen bietet seinen Kundinnen und Kunden (im folgenden Kunden genannt) im Internet ein Onlineportal, über das sie Bahntickets und Zusatzangebote buchen können. Das Portal stellt folgende Funktionalitäten bereit:
Benutzerinnen und Benutzer (im folgenden Benutzer genannt) können sich mit einem Kundenkonto anmelden. Bei der ersten Anmeldung eines neuen Benutzers erfolgt eine Registrierung. Alternativ kann man sich auch als Gast anmelden. Benutzer können eine Bahnverbindung suchen und ein Ticket kaufen. Beim Kauf eines Tickets muss der Benutzer den Zahlungsvorgang über eine der angebotenen Zahlungsschnittstellen aktivieren. Nach erfolgreicher Bezahlung kann der Benutzer das Ticket in digitaler Form auf sein Smartphone laden. Alternativ verschickt der Ticketservice des Unternehmens Tickets auch in gedruckter Form. Der Benutzer kann zu seinem Ticket einen Sitzplatz im Zug reservieren.
Modellieren und zeichnen Sie ein Anwendungsfalldiagramm für das Onlineportal des Bahnunternehmens.

(9 BE)

2.3 Neben dem Online-Portal stellt das Bahnunternehmen native Client-Anwendungen für verschiedene Plattformen zur Verfügung. Ein erstes UML-Klassendiagramm der Client-Server-Architektur liegt in Material 8 vor.

2.3.1 Überführen Sie die Klasse `FahrplanServer` aus dem UML-Klassendiagramm (Material 8) in Anweisungen einer objektorientierten Programmiersprache und implementieren Sie den Konstruktor sowie die Methode `starteServer()`.

Hinweis: Die Dokumentation der Socket-Klassen ist in Material 7 zu finden.

(4 BE)

2.3.2 Der Fahrplan-Client baut eine Socketverbindung zum Fahrplan-Server auf. Nach dem erfolgreichen Verbindungsaufbau sendet der Server eine Meldung an den Fahrplan-Client. Anschließend kann sich der Benutzer als neuer Kunde registrieren lassen oder als Gast beziehungsweise als registrierter Kunde mit E-Mail-Adresse und Passwort anmelden. Nach erfolgreicher Anmeldung wird die Suche nach einer Bahnverbindung angefragt.
Im folgenden Beispiel möchte der Kunde Max Mustermann am 10.06.2022 um 12:00 Uhr von Frankfurt nach Stuttgart fahren. Das Zeichen > steht für die Antwort vom Server, das Zeichen < steht für die Anforderung vom Client, <LF> steht für Line Feed (`'\n'`).

```
> +OK connected<LF>
< login;max.mustermann@xyz.de;Geheim123<LF>
> +OK login<LF>
< Suche Verbindung:Start:Frankfurt,Ziel:Stuttgart,Abfahrt:10.06.2022
12:00:00<LF>
> +OK Verbindung:Frankfurt,ab:ICE 640,10.06.2022 12:20;an:10.06.2022
12:48,Mannheim,ab:ICE 730,10.06.2022 13:05;an:10.06.2022
14:05,Stuttgart;<LF>
< logout<LF>
> +OK logout<LF>
```

- Zur Anmeldung eines Gastes verwendet der Client die Daten des internen Gastkontos (Email: `gast@fahrplan`, Passwort: `inkognito`).
- Die Registrierung eines Neukunden ersetzt die Login-Zeile im Protokoll durch die Anweisung `register;<email>;<passwort><LF>`. Die Antwort des Servers bei erfolgreicher Registrierung lautet: `+OK register<LF>`.

Implementieren Sie die Klasse `FahrplanClient` aus dem UML-Klassendiagramm in Material 8.

Hinweise: Die Methoden `verbinden()`, `anmelden()`, `anmeldenGast()` und `abmelden()` geben den Wert `true` zurück, wenn sie erfolgreich ausgeführt werden konnten. Die Antworten des Fahrplanservers beginnen mit `+OK`, wenn die jeweilige Methodenausführung erfolgreich war. Die Dokumentation der Klasse `DateTime` ist in Material 7 zu finden.

(11 BE)

2.3.3 Die Methode `findeVerbindung()` der Klasse `Fahrplananfrage` ermittelt eine Bahnverbindung und gibt diese als Objekt der Klasse `Verbindung` zurück. Entwickeln und zeichnen Sie ein Objektdiagramm für die Verbindungsinstanz der Anfrage aus Aufgabe 2.3.2.

(6 BE)

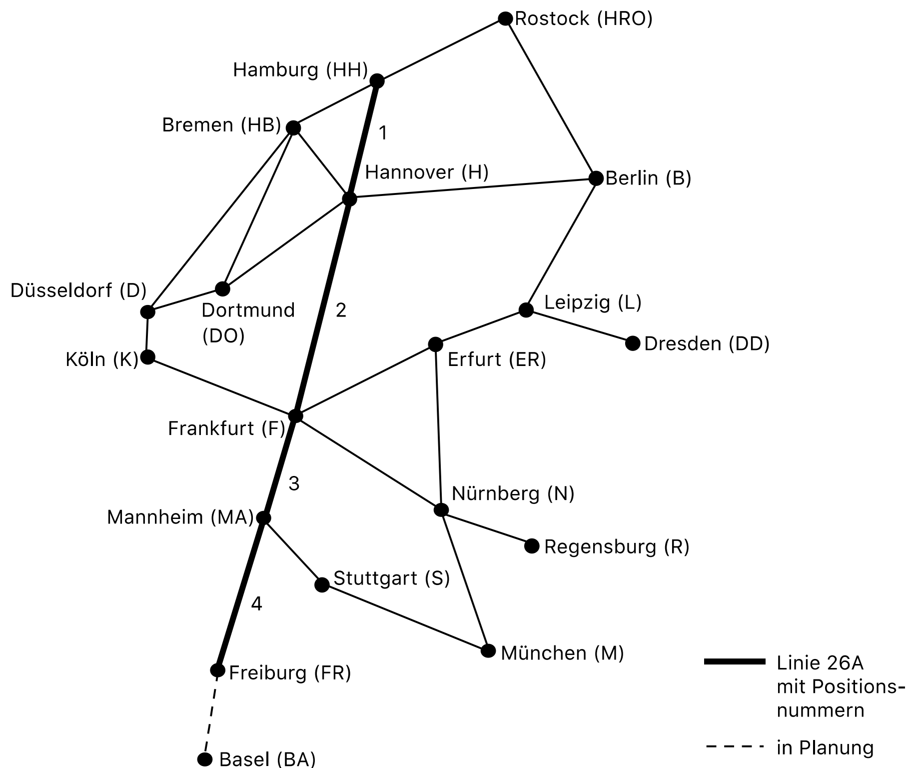
2.3.4 Die Klasse `Fahrplananfrage` (Material 8) bearbeitet Anfragen nach Bahnverbindungen, die der Fahrplan-Client an den Fahrplan-Server sendet. Modellieren und zeichnen Sie den Ablauf der Anfrage für die Bahnverbindung auf Grundlage des Protokolls in Aufgabe 2.3.2 in einem UML-Sequenzdiagramm.

Hinweise: Eine Vorlage für das Sequenzdiagramm ist in Material 9 zu finden. Fehlerüberprüfungen, das Erzeugen und Initialisieren der Haltestationen und Züge sowie das Verarbeiten von Strings sind nicht darzustellen.

(10 BE)

Material 1

Streckennetz



Hinweise:

- Eine Bahnlinie wird identifiziert durch eine Nummer und einen richtungsbezogenen Buchstaben. Zum Beispiel führt die Linie 26A von Hamburg nach Freiburg. Die Linie 26B führt in entgegengesetzter Richtung von Freiburg nach Hamburg.
- Die Segmente einer Linie sind mit Positionsnummern beginnend mit 1 durchnummeriert.

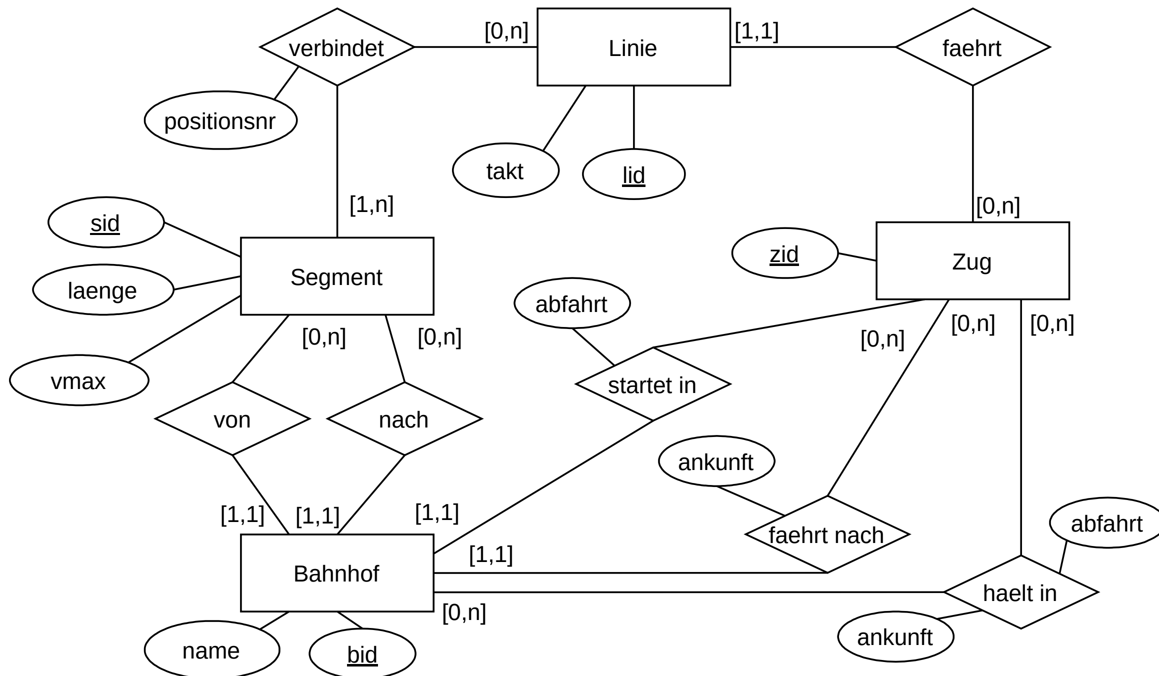
Material 2

Fahrplanausschnitt der Linie 26A

Zug	HH ab	H an	H ab	F an	F ab	MA an	MA ab	FR an
ICE 946	6:45	8:00	8:05	11:02	11:07	11:35	11:40	12:56
ICE 644	7:45	9:00	9:05	12:02	12:07	12:35	12:40	13:56
ICE 944	8:45	10:00	10:05	13:02	13:07	13:35	13:40	14:56
…								

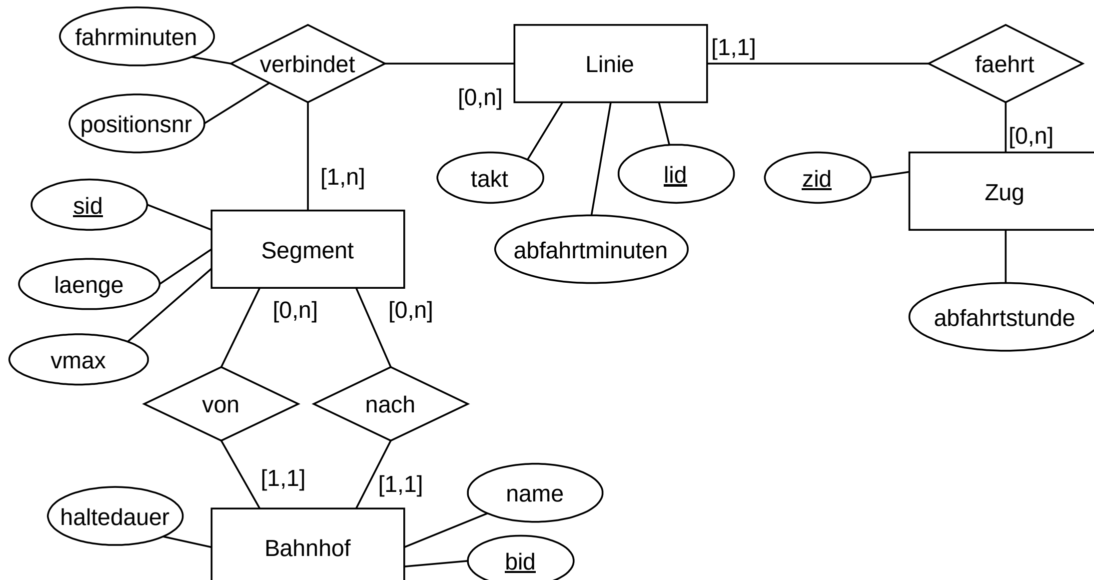
Material 3

Entity-Relationship-Modell A



Material 4

Entity-Relationship-Modell B



Hinweise: Das Attribut `abfahrtstunde` (Entitätstyp `Zug`) speichert die Stunde der Abfahrtszeit. Das Attribut `abfahrtminuten` (Entitätstyp `Linie`) speichert die Minuten der Abfahrtszeit. Für alle Züge einer Linie ist dieser Wert identisch (z.B. Linie 26A in Material 2). Das Attribut `fahrminuten` in der Beziehung `verbindet` speichert die Zeit, die für das jeweilige Streckensegment benötigt wird.

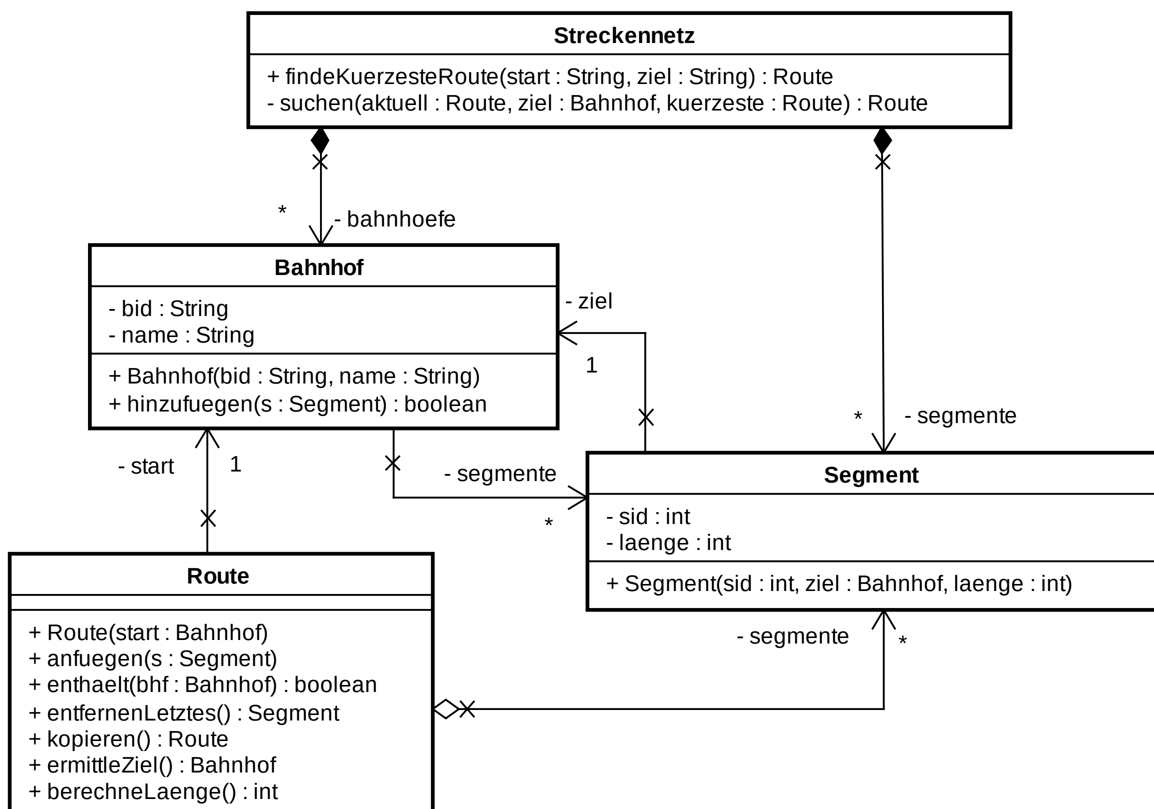
Material 5

Relationenmodell

```
Linie(lid, abfahrtminuten, takt)
Segment(sid, laenge, vmax, von#, nach#)
Verbindung(sid#, lid#, positionsnr, fahrminuten)
Bahnhof(bid, name, haltedauer)
Zug(zid, abfahrtstunde, lid#)
```

Material 6

UML-Klassendiagramm Routenfinder



Hinweise:

Methoden der Klasse `Route`:

- `anfuegen(s: Segment)` fügt ein Segment an das Ende der Liste der `Segmente` an. Die Methode verhindert mehrfaches Hinzufügen desselben Segments.
- `enthaelt(bhf: Bahnhof)` liefert `true`, wenn der Bahnhof in der Route enthalten ist, ansonsten `false`.
- `entfernenLetztes()` entfernt das letzte Segment aus der Route.
- `kopieren()` liefert ein neues Route-Objekt, welches inhaltlich dem Original entspricht.
- `ermittleZiel()` gibt den letzten Bahnhof der Route zurück oder den Startbahnhof, wenn die Route keine Segmente enthält.

Methode der Klasse Bahnhof:

- hinzufuegen(s: Segment) gibt true zurück, wenn das übergebene Segment dem Bahnhof hinzugefügt werden konnte. Andernfalls wird false zurückgeben. Die Methode verhindert mehrfaches Hinzufügen desselben Segments.

Methode der Klasse Streckennetz:

- findeKuerzesteRoute(start: String, ziel: String) sucht die Bahnhöfe mit den Namen start und ziel und ermittelt bei gültigen Namen mithilfe der Methode suchen(aktuell: Route, ziel: Bahnhof, kuerzeste: Route) die kürzeste Route.

Auf alle Attribute kann mit get- und set-Methoden zugegriffen werden.

Material 7

Klassendokumentationen

Klasse List<T>

List<T>

+ List<T>()
+ add(obj : T)
+ add(index : int, obj : T)
+ contains(obj : int) : boolean
+ get(index : int) : T
+ remove(index : int) : T
+ remove(obj : T) : boolean
+ size() : int

List<T>()
erzeugt eine generische Liste mit Elementen des Typs T.

add(obj: T)
hängt das Objekt obj vom Typ T am Ende der Liste an.

add(index: int, obj: T)
fügt das Objekt obj vom Typ T an der Position index in die Liste an.

contains(obj: T): boolean
liefert true, wenn das Objekt obj in der Liste enthalten ist, sonst false.

get(index: int): T
liefert das Listenelement an der Position index zurück bzw. null, falls index negativ oder größer gleich der Anzahl der momentan enthaltenen Elemente ist.

remove(index: int): T
entfernt das Listenelement an der Position index. Liefert das entfernte Element zurück bzw. null, falls index negativ oder größer gleich der Anzahl der momentan enthaltenen Elemente ist.

remove(obj: T): boolean
entfernt das Objekt obj aus der Liste. Falls obj mehrmals in der Liste enthalten ist, wird nur das erste Vorkommen entfernt. Der Rückgabewert ist true, falls das Objekt gefunden und entfernt wurde, sonst false.

size(): int
liefert die Anzahl der Elemente in der Liste zurück.

Klasse ServerSocket

ServerSocket

− localPort : int

+ ServerSocket(localport : int)
+ accept() : Socket
+ close()

ServerSocket(localPort: int)
erzeugt einen Server-Socket und bindet ihn an den angegebenen Port.

accept(): Socket
wartet darauf, dass ein Client eine Verbindung aufbauen will. Wurde eine Verbindung aufgebaut, liefert die Methode das entsprechende Socket-Objekt.

close()
schließt den Server-Socket.

Material 7 (Fortsetzung)

Klasse `Socket`

Socket
− remoteHostIP : String − remotePort : int
+ Socket(remoteHostIP : String, remotePort : int) + connect() : boolean + dataAvailable() : int + read() : int + read(b : byte[], len : int) : int + readLine() : String + write(value : int) + write(b : byte[], len : int) + write(s : String) + close()

`Socket(remoteHostIP: String, remotePort int)`
erzeugt einen Socket-Objekt.

`connect(): boolean`
stellt eine Verbindung zum RemoteHost her; liefert `true`, wenn eine Verbindung hergestellt werden konnte.

`dataAvailable(): int`
liefert die Anzahl der Bytes, die vom Socket gelesen werden können, ohne beim nächsten Aufruf von `read()` zu blockieren.

`read(): int`
liest ein Byte (0..255) vom Socket, bzw. liefert –1, wenn der Socket nicht geöffnet ist. Die Methode blockiert, bis ein Byte verfügbar ist.

`read(b: byte[], len: int): int`
liest bis zu `len` Bytes vom Socket in ein Byte-Array; die Anzahl der tatsächlich gelesenen Bytes wird zurückgeliefert, bzw. –1, wenn der Socket nicht geöffnet ist. Die Methode blockiert, bis mindestens ein Byte verfügbar ist.

`readLine(): String`
liest eine Zeile vom Socket, bzw. liefert `null`, wenn der Socket nicht geöffnet ist. Eine Zeile wird durch ein Zeilenendezeichen abgeschlossen; das Zeilenendezeichen wird jedoch nicht in den zurückgegebenen String übernommen. Die Methode blockiert, bis eine komplette Zeile eingelesen ist.

`write(value: int)`
schreibt ein Byte (0..255) auf den Socket; ist keine Verbindung hergestellt, geschieht nichts.

`write(b: byte[], len: int)`
schreibt `len` Bytes zum Socket; ist keine Verbindung hergestellt, geschieht nichts.

`write(s: String)`
schreibt einen String zum Socket; ist die Schnittstelle nicht geöffnet, geschieht nichts.

`close()`
löst die Verbindung auf.

Klasse `DateTime`

DateTime
+ DateTime() + DateTime(date : String, time : String) + toString() : String

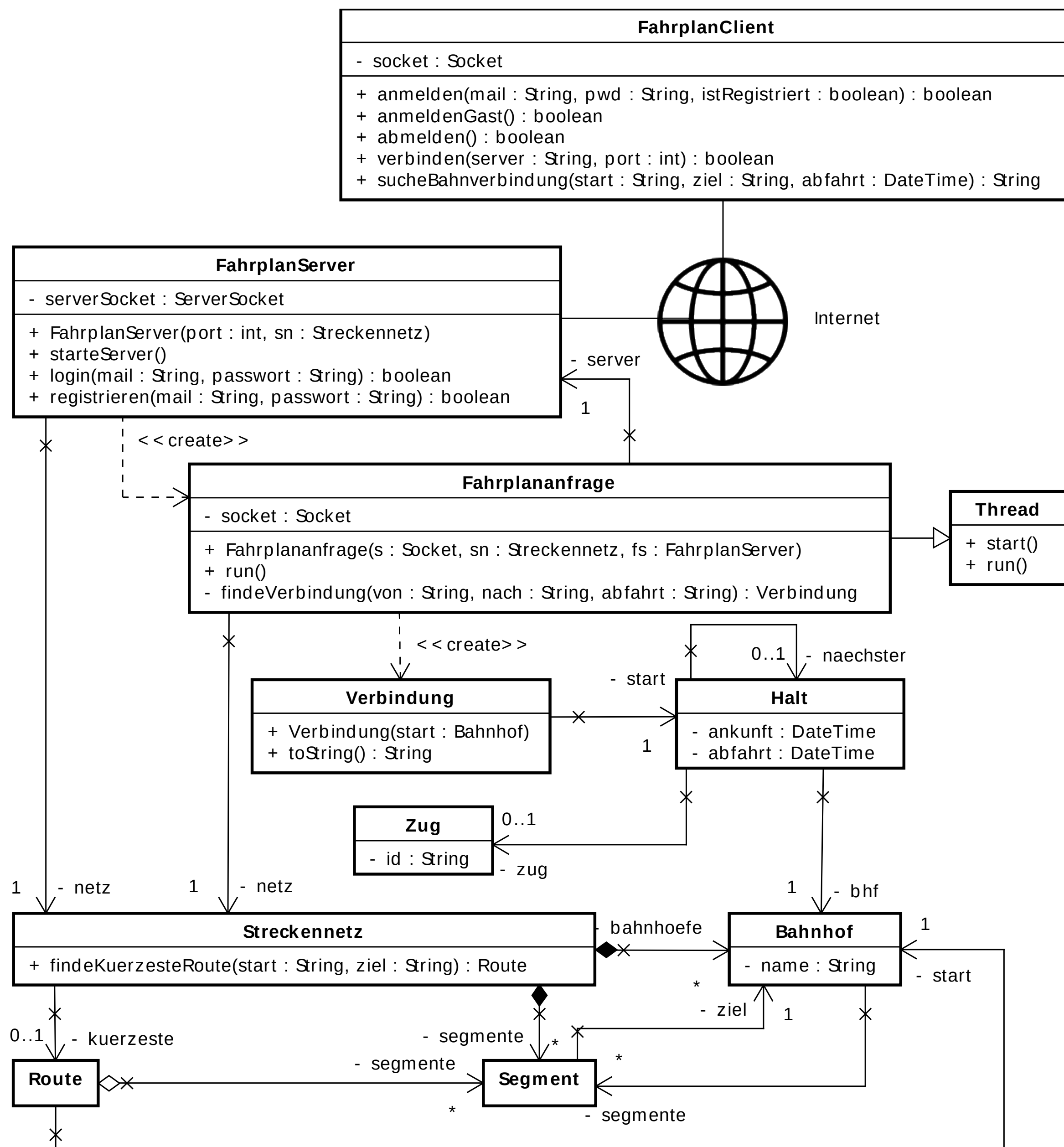
`DateTime()`
erzeugt ein `DateTime`-Objekt mit dem aktuellen Systemdatum und der aktuellen Systemzeit.

`DateTime(date: String, time: String)`
erzeugt ein `DateTime`-Objekt aus einem String mit dem Format `"dd.mm.yyyy"` und einem String mit dem Format `"hh:mm"`.

`toString(): String`
liefert eine String-Repräsentanz des Date-Objekts im Format `"dd.mm.yyyy hh:mm"`.

Material 8

UML-Klassendiagramm Client-Server-Anwendung



Hinweise:

- Die Methode `findeVerbindung(von: String, nach: String, abfahrt: String)` der Klasse `Fahrplananfrage` nimmt die Namen der Bahnhöfe und die Abfahrtszeit als `String` entgegen. Sie sucht im Streckennetz nach der kürzesten Route und erstellt daraus eine Verbindung. Die Abfahrts- und Ankunftszeiten der einzelnen Haltestationen sowie die Züge, die die Verbindung nutzt, werden aus der Datenbank erfragt. Der Datenbankzugriff ist in diesem Klassendiagramm nicht abgebildet.
- Die Methode `toString()` der Klasse `Verbindung` liefert die protokollkonforme Textdarstellung einer Verbindung.

Material 9

Vorlage UML-Sequenzdiagramm

